



# 本科生实验报告

实验课程：\_\_\_\_\_操作系统原理实验\_\_\_\_\_

实验名称：\_\_\_\_\_进程\_\_\_\_\_

专业名称：\_\_\_\_\_计算机科学与技术\_\_\_\_\_

学生姓名：\_\_\_\_\_张玉瑶\_\_\_\_\_

学生学号：\_\_\_\_\_23336316\_\_\_\_\_

实验地点：\_\_\_\_\_实验楼 B203\_\_\_\_\_

实验成绩：\_\_\_\_\_

报告时间：\_\_\_\_\_2025 年 6 月 2 日\_\_\_\_\_

## Section 1 实验概述

- 实验任务 1：编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。
  - 1) 展现系统调用执行结果的正确性，结果截图并说说你的实现思路。
  - 2) 请根据 gdb 来分析执行系统调用后的栈的变化情况。
  - 3) 请根据 gdb 来说明 TSS 在系统调用执行过程中的作用。
- 实验任务 2：实现 fork 函数，并回答以下问题。
  - 1) 请根据代码逻辑和执行结果来分析 fork 实现的基本思路。
  - 2) 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从 fork 返回，根据 gdb 来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完 ProgramManager::fork 后的返回过程的异同。
  - 3) 请根据代码逻辑和 gdb 来解释 fork 是如何保证子进程的 fork 返回值是 0，而父进程的 fork 返回值是子进程的 pid。
- 实验任务 3：实现 wait 函数和 exit 函数，并回答以下问题。
  - 1) 请结合代码逻辑和具体的实例来分析 exit 的执行过程。
  - 2) 请分析进程退出后能够隐式地调用 exit 和此时的 exit 返回值是 0 的原因。
  - 3) 请结合代码逻辑和具体的实例来分析 wait 的执行过程。
  - 4) 如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为 DEAD 开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法。

## Section 2 实验步骤与实验结果

### ----- 实验任务 1 -----

- 任务要求：编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。
  1. 展现系统调用执行结果的正确性，结果截图并说说你的实现思路。

2. 请根据 gdb 来分析执行系统调用后的栈的变化情况。
3. 请根据 gdb 来说明 TSS 在系统调用执行过程中的作用。

- 思路分析：利用 gdb 跟踪。

- 实验步骤：

1. 展现系统调用执行结果的正确性，结果截图并说说你的实现思路。

- 1) 在 syscall.h 中添加新的系统调用函数声明。

```
//第一个系统调用
int syscall_1(int first, int second, int third, int forth, int fifth);
```

- 2) 在 setup.cpp 中实现新的系统调用函数。

```
int syscall_1(int first, int second, int third, int forth, int fifth)
{
    printf("\nhello world, 23336316\n");
    return 0;
}
```

- 3) 在 setup\_kernel()函数中，添加对新系统调用的注册。

```
//设置1号系统调用
systemService.setSystemCall(1, (int)syscall_1);
```

- 4) 在进程 first\_process()中添加新的系统调用。

```
void first_process()
{
    asm_system_call(1);
    asm_halt();
}
```

- 5) 在线程 first\_thread()中运行三次 first\_process()。

```
void first_thread(void *arg)
{
    printf("start process\n");
    programManager.executeProcess((const char *)first_process, 1);
    programManager.executeProcess((const char *)first_process, 1);
    programManager.executeProcess((const char *)first_process, 1);
    asm_halt();
}
```

6) 编译运行。可以看到新加入的系统调用也被成功调用，输出了三行“hello world, 23336316”。

```
Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x2000000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process

hello world, 23336316

hello world, 23336316

hello world, 23336316
```

2、3. 请根据 gdb 来分析执行系统调用后的栈的变化情况并分析 TSS 的作用。

1) 在 first\_process()处打断点，看看系统调用前的栈：

```
38 void first_process()
B+> 39 {
    40     asm_system_call(1);
    41     asm_halt();
    42 }
    43
```

---

```
remote Thread 1.1 In: first_process L39
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8049000 0x8049000
ebp      0x0      0x0
esi      0x0      0
```

x0	0	
eip	0xc0021314	0xc0021314 <first_process()>
eflags	0x202	[ IOPL=0 IF ]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51

从上图中可以看到 cs 寄存器为 0x2b，尾 2 位为 11，即特权级为 3，位于用户态。esp 为 0x804900，是在用户栈。ss 值为 0x3b。

2) 进入 asm\_system\_call，观察中断 int 0x80 执行前的栈：

```
> 146      int 0x80
    147

remote Thread 1.1 In: asm_system_call
eax      0x1      1
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048fb8 0x8048fb8
ebp      0x8048fcc 0x8048fcc
esi      0x0      0
```

x0	0	
eip	0xc00226ed	0xc00226ed <asm_system_call+26>
eflags	0x212	[ IOPL=0 IF AF ]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51

从上图中可以看到 cs 寄存器为 0x2b，尾 2 位为 11，即特权级为 3，位于用户态。esp 为 0x8048fb8，是在用户栈。ss 值为 0x3b。

3) 执行中断 int 0x80，进入 asm\_system\_call\_handler:

```

87  asm_system_call_handler:
> 88      push ds
89      push es

```

remote Thread 1.1 In: asm\_system\_call\_handler L88

```

eax      0x1      1
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xc00256ec 0xc00256ec <PCB_SET+8172>
ebp      0x8048fcc 0x8048fcc
esi      0x0      0

```

```

x0      0
eip      0xc0022697 0xc0022697 <asm_system_call_han
eflags   0x12      [ IOPL=0 AF ]
cs       0x20      32
ss       0x10      16
ds       0x33      51
es       0x33      51

```

```

(gdb) print /x tss
$3 = {backlink = 0x0, esp0 = 0xc0025700, ss0 = 0x10, esp1 = 0x0, ss1 = 0x0,
      esp2 = 0x0, ss2 = 0x0, cr3 = 0x0, eip = 0x0, eflags = 0x0, eax = 0x0,
      ecx = 0x0, edx = 0x0, ebx = 0x0, esp = 0x0, ebp = 0x0, esi = 0x0, edi = 0x0,
      es = 0x0, cs = 0x0, ss = 0x0, ds = 0x0, fs = 0x0, gs = 0x0, ldt = 0x0,
      trace = 0x0, ioMap = 0xc003380c}

```

从上图中可以看到 cs 寄存器为 0x20，尾 2 位为 00，即特权级为 0，这意味着切换到了内核态。

此时 CPU 自发地从 TSS 加载 esp0 和 ss0 到 esp 和 ss，切换到内核栈。可以看到 TSS 的 esp0 为 0xc0025700，eps 的值为 0xc00256ec（值不一样但和接近，因为 CPU 把 esp 设置为 TSS.esp0 后立即压入了用户态上下文，导致 esp 值减小，栈向低地址增长），ss 的值也变成了 TSS 的 ss 值 0x10。

4) 中断 0x80 执行完成后：

```
> 148 pop edi
149 pop esi

remote Thread 1.1 In: asm_system_call L148
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048fb8 0x8048fb8
ebp      0x8048fcc 0x8048fcc
esi      0x0      0

x0      0
eip      0xc00226ef 0xc00226ef <asm_system_call+28>
eflags   0x212    [ IOPL=0 IF AF ]
cs       0x2b     43
ss       0x3b     59
ds       0x33     51
es       0x33     51
```

可以看出中断执行完成后，esp 值、cs 值、ss 值又变回了中断前的值，回到了用户栈，特权级变成 3，ss 变成 0x3b。这说明又回到了用户态。

## ----- 实验任务 2 -----

- 任务要求：实现 fork 函数，并回答以下问题。
  1. 请根据代码逻辑和执行结果来分析 fork 实现的基本思路。
  2. 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从 fork 返回，根据 gdb 来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完 ProgramManager::fork 后的返回过程的异同。
  3. 请根据代码逻辑和 gdb 来解释 fork 是如何保证子进程的 fork 返回值是 0，而父进程的 fork 返回值是子进程的 pid。

- 思路分析：利用 gdb 跟踪。
- 实验步骤：

## 1.分析 fork 实现的基本思路。

### 1) 系统调用框架

用户态调用 `fork()` 会触发 `int 0x80` 中断，进入内核态后调用 `syscall_fork()`，最终由 `ProgramManager::fork()` 处理。内核线程无权调用 `fork()`，通过检查 `PCB::pageDirectoryAddress` 是否为 0 来判断。

### 2) 创建子进程

调用 `executeProcess("", 0)` 创建子进程的 PCB，此时子进程尚未复制资源。子进程的 PCB 被加入调度队列，但处于未初始化状态。

### 3) 复制父进程资源

通过 `copyProcess(parent, child)` 复制父进程资源：

i. 复制内核栈：父进程的内核栈位于  $(\text{int})\text{parent} + \text{PAGE\_SIZE} - \text{sizeof}(\text{ProcessStartStack})$ ，直接 `memcpy` 复制到子进程。设置子进程的 `eax=0`，使其 `fork()` 返回 0，父进程返回子进程 PID。

ii. 初始化子进程 PCB：继承父进程的优先级、时间片等属性，设置 `parentPid` 为父进程 PID。

iii. 复制虚拟地址池：复制父进程的虚拟内存管理位图。

iv. 复制页表和物理内存：为子进程分配新的页目录表，复制父进程前 768 项（用户空间映射）。对每个有效的页表项，分配新的物理页，通过内核中转页复制父进程数据到子进程，更新子进程的页表项指向新物理页。

### 4) 子进程启动准备

设置子进程的 `stack` 指向一个伪造的栈帧，使其被调度时跳转到 `asm_start_process`。`asm_start_process` 通过 `iret` 返回到用户态，从 `fork()` 的返回点继续执行。



## 5) 测试逻辑

```
void first_process() {
    int pid = fork(); // 父子进程在此分叉
    if (pid) {
        printf("I am father, fork return: %d\n", pid);
    } else {
        printf("I am child, fork return: %d\n", pid);
    }
}
```

输出：

```
start process
I am father, fork reutrnr: 2
I am child, fork return: 0, my pid: 2
```

2. 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从 `fork` 返回，根据 `gdb` 来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完 `ProgramManager::fork` 后的返回过程的异同。

首先进入 `first_process` 中的 `fork()` 函数中开始跟踪。

```
Breakpoint 1, first_process () at ../src/kernel/setup.cpp:32
```

```
void first_process()
{
    int pid = fork();
}
```

可以发现 `fork()` 返回的是一个系统调用。

```
fork () at ../src/kernel/syscall.cpp:36
```

```
int fork() {
    return asm_system_call(2);
}
```

进入系统调用。进入 `asm_system_call` 之后，跟踪中断的执行过程。

```
asm_system_call () at ../src/utls/asm_utils.asm:130
```

```
int 0x80
```

进入 int 0x80。展示寄存器。可以发现特权级语句变成 0，进入了内核态。

eax	0x2	2
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0xc0025dec	0xc0025dec <PCB_SET+8172>
ebp	0x8048fac	0x8048fac
esi	0x0	0
x0	0	
eip	0xc0022c77	0xc0022c77 <asm_system_call_handler>
eflags	0x16	[ IOPL=0 AF PF ]
cs	0x20	32
ss	0x10	16
ds	0x33	51
es	0x33	51

之后 call 调用具体的系统调用处理函数。eax 的值为 2，那么调用 system\_call\_table[2]即为 syscall\_fork。

```
asm_system_call_handler () at ../src/utils/asm_utils.asm:88
```

```
sti
call dword[system_call_table + eax * 4]
cli
```

可以发现 syscall\_fork() 返回的是本任务最关键的函数 ProgramManager::fork()。

```
int syscall_fork() {
    return programManager.fork();
}
```

```
// 禁止内核线程调用
PCB *parent = this->running;
if (!parent->pageDirectoryAddress)
{
    interruptManager.setInterruptStatus(status);
    return -1;
}
```

fork 是进程的系统调用，因此我们禁止内核线程调用。因为内核线程并没有设置 PCB::pageDirectoryAddress，所以该项为 0。相反，进程有页目录表，所以该项不为 0。因此，我们通过判断 PCB::pageDirectoryAddress 是否为 0 来判断当前执行的是线程还是进程。

调用 ProgramManager::executeProcess 来创建一个子进程

```
// 创建子进程
int pid = executeProcess("", 0);
```

进入 executeProcess 的执行逻辑。我们像创建一个线程一样创建进程的 PCB。打印 pid 可以发现值为 2。

```
int ProgramManager::executeProcess(const char *filename, int priority)
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    // 在线程创建的基础上初步创建进程的PCB
    int pid = executeThread((ThreadFunction)load_process,
                           (void *)filename, filename, priority);
```

```
(gdb) p pid
$2 = 2
```

找到刚刚新建的 PCB，根据 executeThread 的实现，一个新创建的 PCB 总是被放在 allPrograms 的末尾的。

```
// 找到刚刚创建的PCB
PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
```

```
// 创建进程的页目录表
process->pageDirectoryAddress = createProcessPageDirectory();
//printf("%x\n", process->pageDirectoryAddress);

if (!process->pageDirectoryAddress)
{
    process->status = ProgramStatus::DEAD;
    interruptManager.setInterruptStatus(status);
    return -1;
}
```

```
// 创建进程的虚拟地址池
bool res = createUserVirtualPool(process);

if (!res)
{
    process->status = ProgramStatus::DEAD;
    interruptManager.setInterruptStatus(status);
    return -1;
}
```

在创建进程的页目录表和进程的虚拟地址池后，我们完成了进程的创建。

接着函数返回到 ProgramManager::fork()。

```
ProgramManager::fork (this=0xc0033e40 <programManager>) at ../src/kernel/pr
.cpp:358
```

初始化子进程。然后找到刚刚创建的子进程，然后调用 ProgramManager::copyProcess 来复制父进程的资源到子进程中。

```
// 初始化子进程
PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
bool flag = copyProcess(parent, child);
```

我们来研究 fork 实现最关键的部分——资源的复制，即函数 ProgramManager::copyProcess。

第一步，复制进程 0 级栈。

父进程和子进程的 0 级栈位于各自 PCB 的顶部（PAGE\_SIZE - sizeof(ProcessStartStack)）。我们从父进程的栈中复制 ProcessStartStack 到子进程的栈。最后设置子进程的 eax 为 0（这是 fork 的返回值，保证子进程返回的 pid 为 0）。

```
// 复制进程0级栈
ProcessStartStack *childpss =
    (ProcessStartStack *)((int)child + PAGE_SIZE - sizeof(ProcessStartStack));
ProcessStartStack *parentpss =
    (ProcessStartStack *)((int)parent + PAGE_SIZE - sizeof(ProcessStartStack));
memcpy(parentpss, childpss, sizeof(ProcessStartStack));
// 设置子进程的返回值为0
childpss->eax = 0;
```

可以发现操作过后 childpss 和 parentpss 的信息中，除了 eax 都是一模一样的。其中父进程 eax 值为 2，子进程为 0。

```
(gdb) p/x *childpss
$2 = {edi = 0x0, esi = 0x0, ebp = 0x8048fac, esp_dummy = 0xc0025ddc, ebx = 0x0, edx = 0x0, ecx = 0x0,
    eax = 0x0, gs = 0x0, fs = 0x33, es = 0x33, ds = 0x33, eip = 0xc0022ccf, cs = 0x2b, eflags = 0x216,
    esp = 0x8048f98, ss = 0x3b}

(gdb) p /x *parentpss
$1 = {edi = 0x0, esi = 0x0, ebp = 0x8048fac, esp_dummy = 0xc0025ddc, ebx = 0x0, edx = 0x0, ecx = 0x0,
    eax = 0x2, gs = 0x0, fs = 0x33, es = 0x33, ds = 0x33, eip = 0xc0022ccf, cs = 0x2b, eflags = 0x216,
    esp = 0x8048f98, ss = 0x3b}
```

第二步，设置子进程的线程切换栈。

stack 是 线程切换时的栈帧，用于 asm\_switch\_thread 切换上下文。asm\_start\_process 是子进程的入口函数，它会加载子进程的上下文并开始执行。stack[6] 是 asm\_start\_process 的参数，即子进程的 0 级栈（childpss）。

调度器切换到子进程时，会从 child->stack 恢复上下文，并且处理器会将 stack[4]作为 EIP，stack[5]作为返回地址。

```
// 准备执行asm_switch_thread的栈的内容
```

```
child->stack = (int *)childpss - 7;
```

```
child->stack[0] = 0;
```

```
child->stack[1] = 0;
```

```
child->stack[2] = 0;
```

```
child->stack[3] = 0;
```

```
child->stack[4] = (int)asm_start_process;
```

```
child->stack[5] = 0; // asm_start_process 返回地址
```

```
child->stack[6] = (int)childpss; // asm_start_process 参数
```

```
(gdb) p/x child->stack[4]
```

```
$5 = 0xc0022c20
```

```
(gdb) p/x child->stack[6]
```

```
$6 = 0xc0026dbc
```

第三步，复制 PCB 信息。

让子进程继承父进程的 调度属性（优先级、时间片等），设置父子关系（parentPid），复制进程名。

```
// 设置子进程的PCB
```

```
child->status = ProgramStatus::READY;
```

```
child->parentPid = parent->pid;
```

```
child->priority = parent->priority;
```

```
child->ticks = parent->ticks;
```

```
child->ticksPassedBy = parent->ticksPassedBy;
```

```
strcpy(parent->name, child->name);
```

第四步，复制内存分配信息（虚拟地址池 + 页表 + 物理页）。

接下来将父进程的虚拟地址位图、页目录表、页表、物理页内容都复制给子进程，让子进程拥有和父进程相同的用户空间内存布局。

```

// 复制用户虚拟地址池
int bitmapLength = parent->userVirtual.resources.length;
int bitmapBytes = ceil(bitmapLength, 8);
memcpy(parent->userVirtual.resources.bitmap, child->userVirtual.resour

// 从内核中分配一页作为中转页
char *buffer = (char *)memoryManager.allocatePages(AddressPoolType::KE
if (!buffer)
{

```

经过 copyProcess 的一系列复制操作之后，flag 值为 true，说明父进程的资源被成功复制到子进程。

```

(gdb) p flag
$1 = true

```

```

$2 = {stack = 0xc0026da0 <PCB_SET+12192>,
      name = "U\211\345\203\354\030\350[\v\000\000\000\000\000\000\000", status = READY,
      priority = 1, pid = 2, ticks = 10, ticksPassedBy = 0, tagInGeneralList = {
        previous = 0xc0023e2c <PCB_SET+44>, next = 0x0}, tagInAllList = {
        previous = 0xc0024e34 <PCB_SET+4148>, next = 0x0}, pageDirectoryAddress = -1072594,
      userVirtual = {resources = {length = 94199, bitmap = 0xc0119000 "\001"},
        startAddress = 134512640}, parentPid = 1}

```

接下来返回一堆函数，最终返回到 first\_process()。

```

asm_system_call_handler () at ../src/utils/asm_utils.asm:126
asm_system_call () at ../src/utils/asm_utils.asm:148
fork () at ../src/kernel/syscall.cpp:37
first_process () at ../src/kernel/setup.cpp:35

```

来到关键的打印步骤。打印 pid 可以发现 pid 为 2，输出父进程的语句。

```

(gdb) p pid
$5 = 2

```

```
if (pid)
{
    printf("I am father, fork reutrn: %d\n", pid);
}
```

```
I am father, fork reutrn: 2
```

然后执行 `asm_halt()`。显示 `asm_halt` 执行了两次。第二次执行后 `qemu` 又输出了语句，子进程也输出了。

```
asm_halt();
```

```
else
{
    printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running->pid);
}
```

```
I am father, fork reutrn: 2
I am child, fork return: 0, my pid: 2
```

最后对子进程的一些内容和运行过程做一些总结：

子进程的入口函数是 `asm_start_process`。我们已经把它赋值给了 `child->stack[4]`。

```
(gdb) info symbol 0xc0022c20
asm_start_process in section .text
```

```
(gdb) p/x child->stack[4]
$5 = 0xc0022c20
```

当线程切换时，`asm_switch_thread` 会从子进程的 `stack` 中弹出返回地址（即 `asm_start_process`）。之后，`asm_start_process` 从子进程的 0 级栈（`childpss`）恢复寄存器状态，其中 `eip` 指向了和父进程相同的 `fork()` 的返回地址，`esp` 指向了子进程的用户栈。

当通过 `iret` 指令跳转到 `eip` 时，子进程从 `fork()` 返回。



第一次跳转到 asm\_start\_process。

```
../src/utils/asm_utils.asm
39  asm_start_process:
40      ; jmp $
B+ 41      mov eax, dword[esp+4]
42      mov esp, eax
43      popad
44      pop gs;
45      pop fs;
46      pop es;
47      pop ds;
48
> 49      iret
50
51      ; void asm_ltr(int tr)

remote Thread 1.1 In: asm_start_process          L49    PC: 0xc0022c2d
eax          0x0                                0
ecx          0x0                                0
edx          0x0                                0
ebx          0x0                                0
esp          0xc0025dec                        0xc0025dec <PCB_SET+8172>
ebp          0x0                                0x0
esi          0x0                                0

eax          0x0                                0
eip          0xc0022c2d                        0xc0022c2d <asm_start_process+13>
eflags      0x92                                [ IOPL=0 SF AF ]
cs          0x20                                32
ss          0x10                                16
ds          0x33                                51
es          0x33                                51
```

当 first\_process 打印完一次 “I am father,fork return :2” 进入了 asm\_halt(), 此时在 gdb 输入 next 发现再一次跳转到了 asm\_start\_process。

第二次跳到 asm\_start\_process。

```
../src/utls/asm_utls.asm
39  asm_start_process:
40      ;jmp $
B+ 41      mov eax, dword[esp+4]
42      mov esp, eax
43      popad
44      pop gs;
45      pop fs;
46      pop es;
47      pop ds;
48
> 49      iret
50
51      ; void asm_ltr(int tr)

remote Thread 1.1 In: asm_start_process          L49    PC: 0xc0000000
eax          0x0                                0
ecx          0x0                                0
edx          0x0                                0
ebx          0x0                                0
esp          0xc0026dec                        0xc0026dec <PCB_SET+12268>
ebp          0x8048fac                          0x8048fac
esi          0x0                                0

x0           0
eip          0xc0022c2d                        0xc0022c2d <asm_start_process+13>
eflags      0x282                             [ IOPL=0 IF SF ]
cs          0x20                               32
ss          0x10                               16
ds          0x33                               51
es          0x33                               51
```

这次 return 之后回到了 first\_process，再一次打印 pid，发现值变为 0。

```
39      ; jmp $
40      {
> 41      if (pid)
42      {

remote Thread 1.1 In: first_process
mxcsr       0x1f80                            [ IM DM ZM OM UM PM ]
(gdb) n
asm_system_call () at ../src/utls/asm_utls.asm:148
fork () at ../src/kernel/syscall.cpp:37
first_process () at ../src/kernel/setup.cpp:35
(gdb) p pid
$1 = 0
```

qemu 打印子进程语句。

```
start process  
I am father, fork return: 2  
I am child, fork return: 0, my pid: 2
```

可以对子进程和父进程的返回过程进行一下对比：

父进程通过系统调用进入内核态执行 `syscall_fork`，内核为其创建一个子进程的 PCB 并复制内存布局后，父进程从内核态返回到用户态，此时 `eax` 寄存器保存子进程的 PID，父进程从 `fork()` 的调用处继续执行后续代码，表现上如同普通函数返回。

子进程由调度器选中后，通过 `asm_start_process` 汇编入口恢复上下文，从复制的 `childpss` 结构中加载寄存器状态（包括 `eip` 指向父进程 `fork()` 的返回地址，但 `eax` 强制置 0），最后通过 `iret` 指令切换到用户态。子进程从相同的 `fork()` 返回处开始执行，但返回值 (`eax`) 为 0，从而区分父子进程的逻辑分支。

### 3. 请根据代码逻辑和 gdb 来解释 `fork` 是如何保证子进程的 `fork` 返回值是 0，而父进程的 `fork` 返回值是子进程的 `pid`。

`fork()` 调用时，内核为子进程分配新 PID，通过写时复制共享内存。在返回前，内核分别修改父子进程的寄存器：子进程的 `fork()` 返回值设为 0（因子进程无需管理其他进程），父进程的返回值设为子进程的 PID。

子进程返回 0 的过程是通过内核在创建子进程时显式设置其上下文中的 `eax` 寄存器为 0 实现的：在 `copyProcess` 函数中，子进程的 0 级栈（`childpss`）被初始化为父进程的副本后，强制将 `childpss->eax` 赋值为 0；当子进程首次被调度执行时，调度器通过 `asm_start_process` 汇编函数从 `childpss` 恢复寄存器状态，此时 `eax` 被加载为 0，接着通过 `iret` 指令返回到用户态，使得子进程从 `fork()` 的返回处继续执行时，`eax` 自然为 0，从而与父进程（返回子进程 PID）区分开。这一机制完全依赖内核态对子进程上下文的精准初始化，无需运行时判断，是 `fork()` "一次调用，两次返回" 的核心设计。

```
// 复制进程0级栈
ProcessStartStack *childpss =
    (ProcessStartStack *)((int)child + PAGE_SIZE - sizeof(ProcessStartStack));
ProcessStartStack *parentpss =
    (ProcessStartStack *)((int)parent + PAGE_SIZE - sizeof(ProcessStartStack));
memcpy(parentpss, childpss, sizeof(ProcessStartStack));
// 设置子进程的返回值为0
childpss->eax = 0;
```

### ----- 实验任务 3 -----

- 任务要求：实现 wait 函数和 exit 函数，并回答以下问题。
  1. 请结合代码逻辑和具体的实例来分析 exit 的执行过程。
  2. 请分析进程退出后能够隐式地调用 exit 和此时的 exit 返回值是 0 的原因。
  3. 请结合代码逻辑和具体的实例来分析 wait 的执行过程。
  4. 如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为 DEAD 开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法。
- 思路分析：熟悉 exit 和 wait 的操作，设计样例制造僵尸进程，并利用 schedule 解决。
- 实验步骤：

#### 1. 请结合代码逻辑和具体的实例来分析 exit 的执行过程。

1) 实例：second\_thread 显式调用 exit。

```
void second_thread(void *arg) {
    printf("thread exit\n");
    exit(0);
}
```

2) 在 `exit(0)` 中触发三号系统调用。

```
void exit(int ret) {  
    asm_system_call(3, ret);  
}
```

3) CPU 通过中断 `int0x80` 进入内核态，查找系统调用表，调用 `syscall_exit`:

```
int 0x80  
  
call dword[system_call_table + eax * 4]  
  
void syscall_exit(int ret) {  
    programManager.exit(ret);  
}
```

4) 执行 `ProgramManager::exit()`

先标记当前运行线程为 `DEAD` 状态并保存返回值。

```
// 第一步，标记PCB状态为`DEAD`并放入返回值。  
PCB *program = this->running;  
program->retValue = ret;  
program->status = ProgramStatus::DEAD;
```

然后检查当前运行的是进程还是线程，线程不做处理。在此次由于是线程 (`program->pageDirectoryAddress==0`)，跳过了资源释放。

```
(gdb) p program->pageDirectoryAddress  
$1 = 0
```

最后开始调度，切换到下一个可运行线程。

```
// 第三步，立即执行线程/进程调度。  
schedule();
```

后续是当前线程 `PCB` 变成僵尸线程，等待被 `CPU` 回收。然后下一个线程被调度运行。

2. 请分析进程退出后能够隐式地调用 `exit` 和此时的 `exit` 返回值是 0 的原因。

1) 我们可以发现进程 `first_process` 并没有显式调用 `exit`, 但是输出了 “thread exit”, 说明 `exit` 被隐式调用了。下面分析执行过程。

```
if (pid)
{
    printf("I am father\n");
    asm_halt();
}
else
{
    printf("I am child, exit\n");
}
```

```
start process
I am father
thread exit
I am child, exit
```

2) 进程在初始化时在 `load_process()` 中设置了用户栈。我们在进程的 3 特权级栈中的栈顶处 `userStack[0]` 放入 `exit` 的地址, 然后 CPU 会认为 `userStack[1]` 是 `exit` 的返回地址, `userStack[2]` 是 `exit` 的参数。当进程 `first_process` 返回时, CPU 会按照正常函数返回流程从栈中弹出这些预设值。CPU 从栈中弹出返回地址, 得到 `exit` 地址, 接着弹出 `exit` 的返回地址(0)和参数(0)

```
// 设置进程返回地址
int *userStack = (int *)interruptStack->esp;
userStack -= 3;
userStack[0] = (int)exit;
userStack[1] = 0;
userStack[2] = 0;
```

```
(gdb) info symbol 0xc00225a5
exit(int) in section .text
(gdb) p/x userStack[0]
$3 = 0xc00225a5
```

3)当 `exit` 被隐式调用时,一步步跳转进入关键函数 `ProgramManager::exit()`, 在这里我们标记了 PCB 状态和保存返回值。

`ret` 就是 `exit(0)`传入的参数 0, 把 `program->retValue` 设置为 0, 进程状态设置为 `DEAD`。

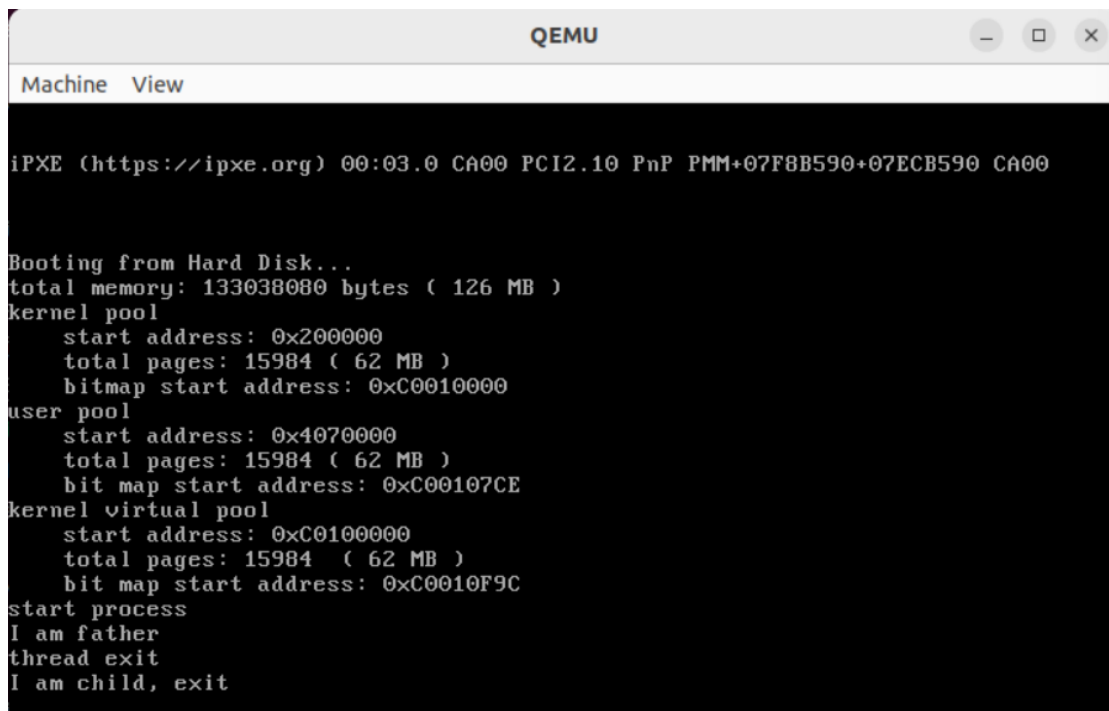
```
// 第一步, 标记PCB状态为`DEAD`并放入返回值。  
PCB *program = this->running;  
program->retValue = ret;  
program->status = ProgramStatus::DEAD;
```

```
(gdb) p program->retValue  
$3 = 0
```

之后进程释放资源, CPU 开始进程调度。

4) 综上所述, 进程退出后能够隐式调用 `exit` 是因为内核在创建进程时预先在其用户栈顶部设置了 `exit` 函数的地址和参数(0), 当进程主函数执行完毕后, CPU 会按照正常函数返回流程从栈中弹出这些预设值, 从而自动跳转到 `exit` 并传入返回值 0。

#### ● 实验结果展示:



```
QEMU  
Machine View  
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00  
  
Booting from Hard Disk...  
total memory: 133038080 bytes ( 126 MB )  
kernel pool  
  start address: 0x200000  
  total pages: 15984 ( 62 MB )  
  bitmap start address: 0xC0010000  
user pool  
  start address: 0x4070000  
  total pages: 15984 ( 62 MB )  
  bit map start address: 0xC00107CE  
kernel virtual pool  
  start address: 0xC0100000  
  total pages: 15984 ( 62 MB )  
  bit map start address: 0xC0010F9C  
start process  
I am father  
thread exit  
I am child, exit
```

### 3. 请结合代码逻辑和具体的实例来分析 wait 的执行过程。

1) 在 first\_process 中设置实例并测试。父进程调用 wait。

```
void first_process()
{
    int pid = fork();
    int retval;

    if (pid)
    {
        pid = fork();
        if (pid)
        {
            while ((pid = wait(&retval)) != -1)
            {
                printf("wait for a child process, pid: %d, return value: %d\n",
                    pid, retval);
            }

            printf("all child process exit, programs: %d\n",
                programManager.allPrograms.size());

            asm_halt();
        }
        else
        {
            uint32 tmp = 0xffffffff;
            while (tmp)
                --tmp;
            printf("exit, pid: %d\n", programManager.running->pid);
            exit(123934);
        }
    }
    else
    {
        uint32 tmp = 0xffffffff;
        while (tmp)
            --tmp;
        printf("exit, pid: %d\n", programManager.running->pid);
        exit(-123);
    }
}
```

```
int wait(int *retval) {
    return asm_system_call(4, (int)retval);
}
```

```
int syscall_wait(int *retval) {
    return programManager.wait(retval);
}
```

2) 进入 ProgramManager::wait() 执行流程。



第一步，查找子进程。试图在 allPrograms 中找到一个状态为 DEAD 的子进程。allPrograms 中包含了所有状态的所有进程和线程。

```
item = this->allPrograms.head.next;

// 查找子进程
flag = true;

while (item)
{
    child = ListItem2PCB(item, tagInAllList);
    if (child->parentPid == this->running->pid)
    {
        flag = false;
        if (child->status == ProgramStatus::DEAD)
        {
            break;
        }
    }
    item = item->next;
}
```

接下来找到了一个可回收的子进程。当 retval 不为 nullptr 时，我们取出子进程的返回值放入到 retval 指向的变量中。然后取出子进程的 pid，调用 releasePCB 来回收子进程的 PCB。

```
if (item) // 找到一个可返回的子进程
{
    if (retval)
    {
        *retval = child->retValue;
    }

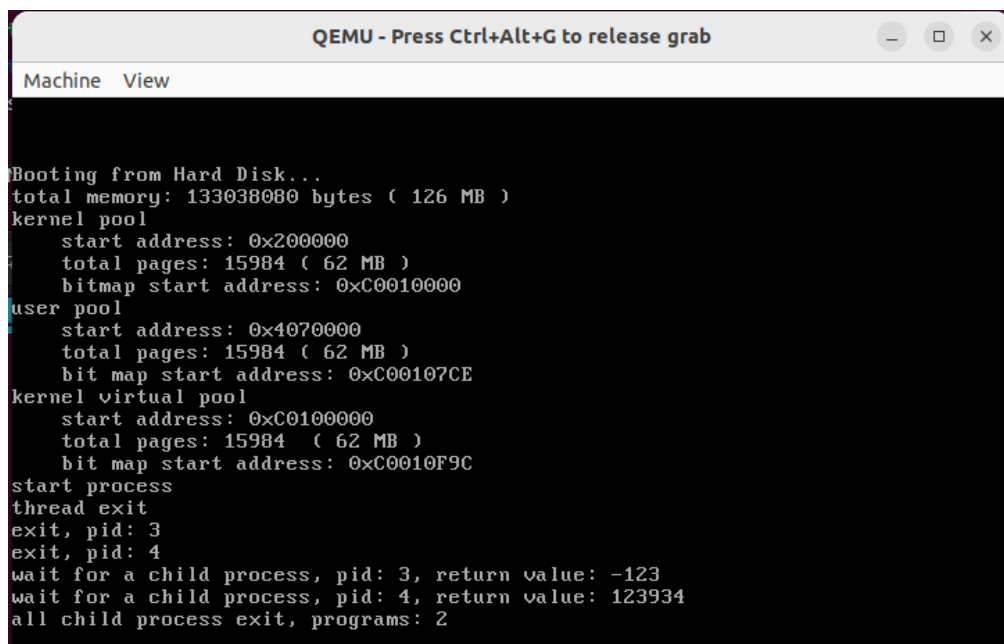
    int pid = child->pid;
    releasePCB(child);
    interruptManager.setInterruptStatus(interrupt);
    return pid;
}
```

```

else
{
    if (flag) // 子进程已经返回
    {
        interruptManager.setInterruptStatus(interrupt);
        return -1;
    }
    else // 存在子进程，但子进程的状态不是DEAD
    {
        interruptManager.setInterruptStatus(interrupt);
        schedule();
    }
}
}

```

3) 结果展示。



The screenshot shows a QEMU window titled "QEMU - Press Ctrl+Alt+G to release grab". The terminal output displays the following information:

```

Machine  View
Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
thread exit
exit, pid: 3
exit, pid: 4
wait for a child process, pid: 3, return value: -123
wait for a child process, pid: 4, return value: 123934
all child process exit, programs: 2

```

4.如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为 **DEAD** 开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法。

1) 设计测试样例制造僵尸/孤儿进程。

在 `first_process` 中创建 3 个子进程。

在样例中故意只 wait 两次，导致有一个子进程得不到 wait，从而变成僵尸进程。

```
void first_process()
{
    int pid=-1;
    // 创建3个子进程
    for (int i = 0; i < 3; i++) {
        pid = fork();

        if (pid > 0)    //父进程分支
            continue; //继续创建下一个子进程
        else if (pid == 0) { //子进程分支
            //产生耗时，使父进程提前退出制造僵尸进程
            int workload=(i==0)?10000:10; //第一个子进程延迟更大

            for (int j = 0; j < workload; j++) {
                for (int k = 0; k < workload / 1000; k++);
            }

            printf("\nextit, pid: %d\n", programManager.running->pid);
            exit(888);
        }
    }

    int exit_code;

    int count=2;
    while ((pid = wait(&exit_code)) != -1 && count) {
        printf("wait for a child process, pid: %d, return value: %d\n", pid, exit_code);
        count--;
    }

    printf("all child process exit, programs: %d\n", programManager.allPrograms.size());
    asm_halt();
}
```

## 2) 修改 schedule 函数。

在 schedule 中加入如图所示内容：当一个进程执行完后，如果状态设置为 DEAD，检查父进程是否存在：如果父进程存在，打印“find father”并退出检查操作；否则进程自行释放 PCB，完成回收操作。

```

else if (running->status == ProgramStatus::DEAD)
{
    if(!running->pageDirectoryAddress) {
        releasePCB(running);
    }
    else{
        int temp = 0;
        ListItem* item = this->allPrograms.head.next;
        int fatherpid=running->parentPid;
        PCB* father;

        while (item)
        {
            father = ListItem2PCB(item, tagInAllList);
            if (father->pid == fatherpid)
            {
                temp = 1;
                printf("find father \n");
                break;
            }
            item = item->next;
        }

        if(!temp){
            releasePCB(running);
        }

        }printf("scheduling... now have %d program\n",programManager.allPrograms.size());
    }
}

```

3) 测试运行。可以发现三个进程退出时都出现了“find father”，说明退出时父进程仍在。后面调用了两次 wait，分别释放了 3，4。可以看到在释放 wait 前一共有 5 个进程，wait 之后父进程结束输出“all child process exit”，此时显示只有两个进程，子进程完全被回收。说明僵尸进程 pid=5 也被回收了。

```

start process
new process is forked! pid:3, now have 4 program
new process is forked! pid:4, now have 5 program
new process is forked! pid:5, now have 6 program

second thread exit
exit, pid: 2
scheduling... now have 5 program

exit, pid: 3
find father
scheduling... now have 5 program

exit, pid: 4
find father
scheduling... now have 5 program

exit, pid: 5
find father
scheduling... now have 5 program
wait for a child process, pid: 3, return value: 000
wait for a child process, pid: 4, return value: 000
all child process exit, programs: 2

```

## Section 5 实验总结与心得体会

本次学习主要学习了进程之间的相关机制。我学习了跟踪子进程的创建，僵尸进程和孤儿进程的回收，`exit` 和 `wait` 的实现逻辑，更进一步地加深了我对操作系统设计的理解。期间当然也遇到不少困难，比如说错误理解了僵尸进程和孤儿进程的概念，没理解透彻 `wait` 的逻辑，走了不少弯路，好在最后一一克服，收获颇丰。